OPRA FAST decoder
Overview

- Application Overview
- Compiler Features
  - IO Channels
  - Loop Pipelining
- Kernel Implementation
- Host Implementation
Application Overview
10G OPRA FAST

- 10G Ethernet UDP stream
- Headers removed by OpenCL kernel
- Decode compressed OPRA FAST data
- Reconstruct messages
- Platform with 10 GbE I/O channels and kernels using Altera’s OpenCL
- < .5 uS Latency
## Application structure
- OPRA FAST decoder kernel written in OpenCL
- The decoder outputs the reconstructed OPRA messages
- Output sent through a kernel to kernel channel for subsequent processing

## Modularity
- Users can plug custom trading kernel
- Currently using a dummy trading kernel for verification
OPRA FAST encoding

- Each message contains several compressed fields
- Presence Map of each message tells us which fields of the current message are encoded
  - Other fields are repeated from the last message or incremented
- Compressed fields are variable number of bytes
  - Each byte contains 7 bits of data and ‘stop bit’ for last byte
  - Uncompressed fields are fixed size

SOH Ver.  Sequence Number of 1st Message  # Msgs in Frame

01:02:30:30:30:30:30:31:38:36:33:33:30:30:31:


MsgSz  Presence Map  Msg Type  Fields
Trading algorithm

- **Dummy trading algorithm for verification purposes**
  - Demo goal to demonstrate parsing OPRA packets at line rate
  - Sending all decompressed data back through UDP would bottleneck the processing

- **Return a subset of the fields**
  - Required to throttle the amount of data sent out through UDP
  - Host configures which field range is returned
Compiler Features
Altera I/O Channels

- Allows kernels to interface to outside world
  - Simple API to read and write data from external sources
  - Available channel are board-specific, defined by board designer

- This example uses IO channels connected to a UDP Offload Engine to communicate over 10 Gbps Ethernet

```c
kernel void foo()
{
    read_channel_altera();
    process_data();
    write_channel_altera();
}
```
Loop Pipelining: Loop Carried Dependencies

- Loop-carried dependencies are dependencies where one iteration of the loop depends upon the results of another iteration of the loop.

```c
__kernel void generate_rngs(ulong num_rnds) {
    t_state_vector state = initial_state();
    for (ulong i=0; i<num_rnds; i++) {
        state = generate_next_state( state );
        unit y = extract_random_number( state );
        write_channel_altera(RANDOM_STREAM, y);
    }
}
```

- The variable state in iteration 1 depends on the value from iteration 0. Similarly, iteration 2 depends on the value from iteration 1, etc.
Loop Pipelining (2)

- To achieve acceleration, we can **pipeline** each iteration of a loop containing loop carried dependencies
  - Analyze any dependencies between iterations
  - Schedule these operations
  - Launch the next iteration as soon as possible

```c
__kernel void generate_rngs(ulong num_rnds)
{
    t_state_vector state = initial_state();
    for (ulong i=0; i<num_rnds; i++) {
        state = generate_next_state( state );
        unit y = extract_random_number( state );
        write_channel_altera(RANDOM_STREAM, y);
    }
}
```

At this point, we can launch the next iteration.
Loop Pipelining Example

- No Loop Pipelining

- With Loop Pipelining

No Overlap of Iterations!

Finishes Faster because Iterations Are Overlapped

Looks almost like ND-range thread execution!
Kernel Implementation
**OPRA Field Parser**

- **UDP packet split into fixed length frames**
  - OPRA FAST processing expressed as a loop
  - Allow the OpenCL compiler to extract pipeline parallelism from loop iterations
  - Each iteration processes one frame

- **Fields may span across multiple frames**
  - Loop carried dependencies
  - The compiler understands and generates efficient hardware in the presence of dependencies

---

**UDP IP**

Pipelined parallel processing of frames
OPRA Decoding Loop

- Every loop iteration, 8 byte frame of data is read from the UDP interface
  - Except in rare case when there is data left from last frame

- The multiple fields of the OPRA message are constructed from over several iterations of the loop
  - As each field is decoded, it is written to a location in the message structure

- When the message is fully parsed, it is sent to the trading kernel via a kernel-to-kernel channel
  - Non-present fields (according to the Presence Map) are maintained from last message or incremented

- Loop carried dependencies are minimized to allow one iteration to launch every cycle
  - If hardware frequency $\geq 156.25$ MHz, we can saturate the 10G connection
Other Kernels

- UDP IO data interface is 16 bytes wide, while the Decoder kernel takes 8 bytes, two kernels are used as 16-to-8 byte adaptors
- Two kernels are used to choose between sending data from the IO channels or from global memory
Latency measurement

Kernel runs at 192 MHz

10G XCVR → UDP IP → 10G UDP → OPRA Decoder

UDP stack latency
OPRA decoder latency

Decoded messages
IO Channel
MAC+ UDP interface

25 cycles 38 cycles
0.328 μs
Host Implementation
Host Program

- OPRA Kernels can either communicate over UDP IO Channels, or by reading and writing to memory
- If UDP IO channels are being used, the host will send and receive data over UDP sockets to the IP address of the FPGA card
  - 10G Ethernet card should be installed in host PC, connected to FPGA card
- The host program forks into two processes, which allows the host to send and receive data over UDP independently
- Tested using Solarflare network interface card
  - OpenOnload drivers are used to accelerate UDP transfers, and are needed to consistently saturate the 10G interface
Tips for tuning host to achieve line rate

- **Host OS is not a real time OS**
  - System jitter can cause packet loss on the host
  - Do not run any unnecessary services or applications

- **Run app through demo.sh script**
  - Does some driver tuning to minimize overhead
  - This proves sufficient on the test machine we have on our side
    (Intel(R) Core(TM)2 Quad CPU  Q9550  @ 2.83GHz)
Running the example design

host/opra [mode] [UDP frames]

- [mode] configures where the data source and destination are located
  - memory input → memory output (default, 0)
  - UDP input → memory output (1)
  - Memory input → UDP output (2)
  - UDP input → UDP output (3)

- [UDP frames] specifies how many frames to transmit
  - The frames are read from a pcap file that comes with the example
Demo output

> host/opra 3 300000

Using AOCX: opera.aocx
Short memory run --> results.txt
Long run, use as reference for subsequent runs
Use compact UDP packets for better throughput
All integrity checks are DONE.

Performance testing: UDP Rx --> OPRA decoder --> UDP Tx

- Verified field 0
- Verified field 1
- Verified field 2
- Verified field 3
- Verified field 4

Memory based runs for integrity checks and generating reference data

OPRA stream throughput: 9.417523 Gbit / s
(99.7 % of theoretical maximum of 9.446Gb/s)

Multiple UDP runs
In each run, configure the dummy trading algo to return one field of the message

Max performance measured across all runs
Thank You